Snakes and Sandwiches: Optimal Clustering Strategies for a Data Warehouse

H. V. Jagadish*
U of Illinois, Urbana-Champaign
jag@cs.uiuc.edu

Laks V. S. Lakshmanan†
IIT – Bombay
laks@math.iitb.ernet.in

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

Abstract

Physical layout of data is a crucial determinant of performance in a data warehouse. The optimal clustering of data on disk, for minimizing expected I/O, depends on the query workload. In practice, we often have a reasonable sense of the likelihood of different classes of queries, e.g., 40% of the queries concern calls made from *some* specific telephone number in *some* month. In this paper, we address the problem of finding an optimal clustering of records of a fact table on disk, given an expected workload in the form of a probability distribution over query classes.

Attributes in a data warehouse fact table typically have hierarchies defined on them (by means of auxiliary dimension tables). The product of the dimensional hierarchy levels forms a lattice and leads to a natural notion of query classes. Optimal clustering in this context is a combinatorially explosive problem with a huge search space (doubly exponential in number of hierarchy levels). We identify an important subclass of clustering strategies called *lattice paths*, and present a dynamic programming algorithm for finding the optimal lattice path clustering, in time linear in the lattice size. We additionally propose a technique called *snaking*, which when applied to a lattice path, always reduces its cost. For a representative class of star schemas, we show that for every workload, there is a snaked lattice path which is globally optimal. Further, we prove that the clustering obtained by applying snaking to the optimal lattice path is never much worse than the globally optimal snaked lattice path clustering. We complement our analyses and validate the practical utility of our techniques with experiments using TPC-D benchmark data.

* This work was largely performed when the author was at AT&T Labs–Research, Florham Park, NJ 07932, USA.

† Currently on leave from Concordia University, Canada. This work was largely performed when the author was visiting AT&T Labs–Research, Florham Park, NJ 07932, USA.

1 Introduction

There is tremendous current interest in data warehousing and OLAP applications. OLAP applications typically view data as having multiple logical *dimensions* (e.g., product, location), with natural hierarchies defined on each dimension, and analyze the behavior of various *measure attributes* (e.g., sales, volume) in terms of the dimensions. Such an organization is called a *star schema*. OLAP queries typically involve selections and groupbys on certain dimensions of the star schema, often aggregating measure attributes, over a very large number of tuples in the fact table.

Viewing a fact table together with its various dimension tables as a multi-dimensional grid, we call a vector of (dimension, value) pairs a *grid query*. When all the values are from the leaf levels of the associated dimension hierarchies, the grid query corresponds to an individual cell in the multi-dimensional grid. When one or more values are from higher up in their dimension hierarchies, the query corresponds to subgrids (rectangles in two dimensions — see Figure 1). The result returned by a grid query could be all the selected tuples, an aggregate (e.g., *sum*), or some other function; this detail does not concern us. In our experience, almost all data analysis queries issued against a data warehouse are grid queries. Even a typical OLAP session involving operations such as `cube`, `rollup`, and `drilldown`, repeatedly invokes various grid queries.

Database performance, particularly for data intensive OLAP queries, is largely determined by the cost of I/O required to process each query. This I/O, in turn, depends on how records are physically laid out on disk. We know that there can be no linear clustering of records that will permit all queries over a multi-dimensional space to be answered efficiently. However, given a query workload, it is possible to define an optimal clustering of records on disk that minimizes the expected I/O cost. For realistic data warehouses, the number of possible grid queries (which is the product of the sizes of the hierarchies in each dimension) is likely to be extremely large, even compared to the

number of queries issued over a long time period. This makes the obtaining of *stable* workloads in terms of the distribution of individual queries extremely hard, if not impossible. Fortunately, there is a viable alternative, that of specifying workloads in terms of query classes, instead of in terms of individual queries.

We define the notion of a (*grid*) *query class* based on the levels associated with the dimension values defining the grid query. Since the number of query classes (which is the product of the number of levels in each dimension hierarchy) is likely to be several orders of magnitude smaller than the number of possible individual queries, statistics compiled over the query stream can be used to obtain a fairly good and stable characterization of the distribution of queries across query classes. For instance, a data warehouse administrator may know that 30% of the queries ask about sales of jeans by type of jeans across *some* state; an additional 25% of the queries may ask about overall sales of jeans by *individual* city; and so forth (see Figure 1). One can now address the problem of obtaining an optimal disk clustering of records for this notion of a workload. More precisely, we address the following problem:

> Given an anticipated workload in terms of the frequencies of queries belonging to different query classes, how can one exploit this information to obtain (efficiently) a good clustering that minimizes the expected disk I/O cost?

Our contributions in this paper are as follows:

- We define an important class of clustering strategies called (*monotone*) *lattice paths*, for which the optimal strategy can be computed efficiently using a dynamic programming algorithm (Sections 3 and 4). Our algorithm is linear in the size of the lattice of query classes, and quadratic in the number of dimensions.

- We introduce an improvement called *snaking*, which is very cheap to compute for a given lattice path, and which when applied to a lattice path *always* reduces the expected I/O cost, over all workloads (Section 5).

  We analytically establish that for the case of two dimensions with complete binary tree hierarchies, *the desired global optimal strategy is always some snaked lattice path*. Our proof technique suggests this is likely to be the case in general. We also prove that the clustering obtained by applying snaking to the optimal lattice path has an expected cost within a factor of 2 of the optimal snaked lattice path.

- We complement our analyses and validate the practical utility of our techniques with results obtained from experiments using TPC-D benchmark data (Section 6).

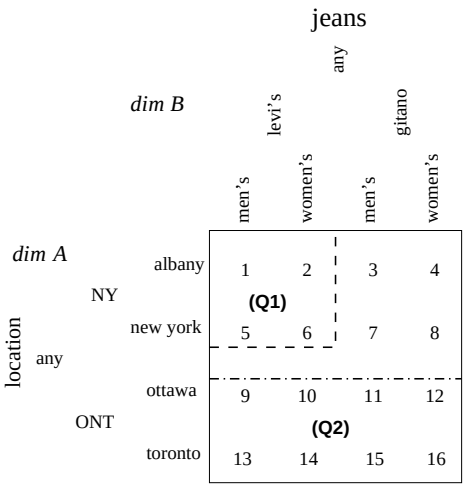

Figure 1: Example Dimension Hierarchies on Two-Dimensional Sales Data with Row Major Clustering

For lack of space, we only sketch the proofs of some results and suppress the others, all of which are discussed in detail in [14].

2 Motivating Example

A point that cannot be emphasized enough is that the choice of clustering strategy can make orders of magnitude difference in I/O cost, and hence in query performance. The problem we are seeking to address is one in which the benefit obtained can be significant.

Example 1 [A Typical (Toy) Data Warehouse] Consider the schema of Figure 1, with relations:

```
location(state, city, lid)
jeans(type, gender, jid)
sales(lid, jid, sale)
```

Typical OLAP queries are given below, where `type` has the domain {levi's, gitano}, `gender` has the domain {men's, women's}, etc.

```
Q1: select   sum(sales)
    from     sales, location, jeans
    where    sales.lid = location.lid and
             sales.jid = jeans.jid and
             location.state = NY and
             jeans.type = levi's

Q2: select   location.city, jeans.type,
             sum(sales)
    from     sales, location, jeans
    where    sales.lid = location.lid and
```

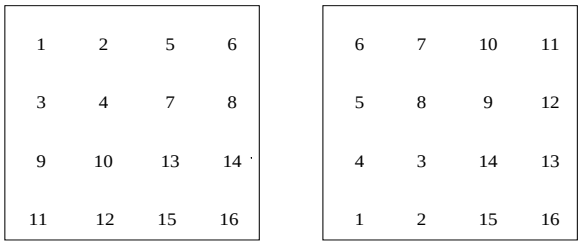

Figure 2: Some More Possible Clustering Strategies: (a) Quadrant Based (Z Curve), (b) Hilbert Curve.

```
        sales.jid = jeans.jid and
        location.state = ONT
group by location.city, jeans.type
```

The first query simply fetches all tuples corresponding to the sale of levi's jeans in state NY and performs an aggregate operation on them. Query (`Q2`), on the other hand, performs a selection and a groupby and essentially accesses all tuples corresponding to the sale of jeans in state ONT. Figure 1 shows the fact tuples accessed by (`Q1`) and (`Q2`) pictorially. (`jeans` = levi's, `location` = NY) and (`jeans` = any, `location` = ONT) are the grid queries corresponding to (`Q1`) and (`Q2`), respectively.

Figure 1 shows a simple row major strategy, denoted P_1 in the sequel, for clustering the sales data cells. Figure 2 shows two additional clustering strategies for the same data: strategy (a), denoted P_2, first locally orders the cells inside each (2×2) subgrid in a row major fashion, and then orders the four (2×2) subgrids themselves row major; strategy (b) corresponds to the well-known Hilbert curve [6, 12], denoted H_d^2.

For each expected query, one can compute the cost of evaluating the query given a clustering strategy. We use the number of seeks (non-sequential disk accesses) required as a simple analytical measure of query cost. In doing so, we follow the footsteps of existing literature.

The expected cost, by this measure, is the same for all queries in a query class. As such, we compute costs per query class, and name query classes based on the level in the hierarchy that is selected in each dimension. (The nomenclature for the query classes will be explained in Section 3, immediately after this example). For example, in Figure 1, the grid query (`jeans` = men's levi's jeans, `location` = toronto), corresponding to one cell, is in class $(0, 0)$; whereas the query (`jeans` = any, `location` = any), corresponding to the whole grid, is in class $(2, 2)$.

Table 1 shows the average (over all queries in the query class) cost of the various query classes under each clustering strategy. In each table entry, the average cost is written in the form ⟨total cost over all queries in class⟩/⟨number of queries in the query class⟩. The last two columns show the costs for the snaked versions $\widetilde{P_1}$ and $\widetilde{P_2}$, of the lattice paths P_1 and P_2, and will be discussed in Section 5.

Query Class	Strategy				
	P_1	P_2	H_d^2	$\widetilde{P_1}$	$\widetilde{P_2}$
(0,0)	16/16	16/16	16/16	16/16	16/16
(1,1)	8/4	4/4	4/4	6/4	4/4
(2,2)	1/1	1/1	1/1	1/1	1/1
(1,0)	16/8	16/8	10/8	14/8	12/8
(0,1)	8/8	8/8	10/8	8/8	8/8
(2,0)	16/4	16/4	8/4	13/4	12/4
(0,2)	4/4	8/4	9/4	4/4	6/4
(2,1)	8/2	4/2	2/2	5/2	3/2
(1,2)	2/2	2/2	3/2	2/2	2/2

Table 1: Average Query Class Cost

Workload	Strategy				
	P_1	P_2	H_d^2	$\widetilde{P_1}$	$\widetilde{P_2}$
1	17/9	15/9	49/36	14/9	25/18
2	13/6	11/6	31/24	21/12	9/6
3	1	5/4	3/2	1	9/8

Table 2: Expected Workload Cost

Table 2 shows the expected cost of each strategy for each of the following three workloads:

1. All query classes are equally likely,
2. None of the query classes $(0, 1), (0, 2), (1, 1)$ is likely, and the remaining query classes are equally likely,
3. Only the query classes $(0, 0), (0, 1), (0, 2), (1, 2)$ are likely, and with equal probability.

One can see that even for this very small example, one strategy can easily have close to twice the cost of another. If we increase the grid size, this difference becomes more dramatic. Table 3 shows how the relative costs of the three strategies under the three workloads above vary as we increase the fanout. In each case, we show the savings, in expected cost, of the best strategy (among P_1, P_2, H_d^2) w.r.t. the worst. We observe wide variation in the expected cost of the three strategies for different workloads and fanout. Even with a modest

Workload	fanout = 2	fanout = 4	fanout = 32
1	72%	61%	52%
2	60%	42%	27%
3	67%	30%	0.7%

Table 3: Relative Costs for Varying Fanouts

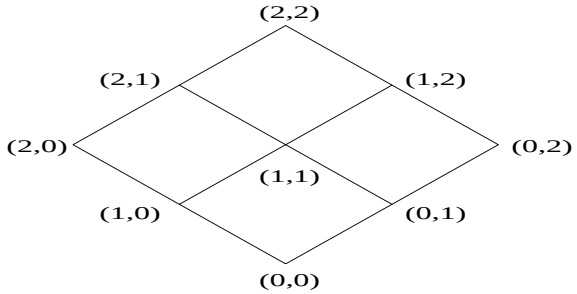

Figure 3: Lattice Associated with the Star Schema of Figure 1. $f(A,i) = f(B,i) = 2,\ 1 \leq i \leq 2$.

fanout of 32 in each of the two levels in either hierarchy over a two-dimensional attribute space, we see well over a factor of 140 difference in cost for workload 3. ■

3 Basic Notions

Let **S** be a star schema with k dimensions, denoted $1, \ldots, k$. For easier comprehension, we will use the letters $A, B, \ldots$ to denote the dimensions, in place of numbers $1, 2, \ldots$, when the number of dimensions is fixed and small. Assume that the hierarchy associated with each dimension is a balanced tree, i.e., the length of a path from the root to any leaf is the same. We will discuss in Section 4.1 how to relax this assumption. We count the levels of the hierarchy from the leaves (level 0) up. Assume that dimension d has a hierarchy with ℓ_d levels, $1 \leq d \leq k$. We let $f(d,i)$, for dimension d and level $1 \leq i \leq \ell_d$, denote the average fanout, at level i, of the hierarchy associated with dimension d.

Definition 1 [Query Class] A *query class* is a k-vector of level numbers of the form $(i_1, ..., i_k)$ such that $0 \leq i_1 \leq \ell_1, ..., 0 \leq i_k \leq \ell_k$. ■

For example, in Figure 1, as explained in Section 1, the grid query (men's levi's jeans, toronto) (which is one cell) is in class $(0,0)$, whereas the query (any, any) (which is the whole grid) is in class $(2,2)$.

Consider the set $\mathcal{L} = \{(i_1, ..., i_k) \mid 0 \leq i_1 \leq \ell_1,\ ...,\ 0 \leq i_k \leq \ell_k\}$ of all query classes, and define a partial order $\leq$ on $\mathcal{L}$ as $(i_1, ..., i_k) \leq (j_1, ..., j_k)$ iff $i_1 \leq j_1, ..., i_k \leq j_k$. It is straightforward to see that $(\mathcal{L}, \leq)$ is a complete lattice, with the least element $\perp = (0, ..., 0)$ and the greatest element $\top = (\ell_1, ..., \ell_k)$. We call $(\mathcal{L}, \leq)$ the *query class lattice* associated with star schema **S**. With this, we can formalize the notion of a workload that we have discussed informally so far.

Definition 2 [Workload] Given a query class lattice $\mathcal{L}$, a *workload* is a probability distribution over the set of all query classes in $\mathcal{L}$. ■

For $\vec{u} = (i_1, ..., i_k)$ and $\vec{v} = (j_1, ..., j_k)$ in $\mathcal{L}$, we say $\vec{v}$ is a *d-successor* of $\vec{u}$ if $j_1 = i_1, ..., j_{d-1} = i_{d-1}, j_d = i_d + 1, j_{d+1} = i_{d+1}, ..., j_k = i_k$. In this case we define the *weight* of the edge $(\vec{u}, \vec{v})$ as $wt(\vec{u}, \vec{v}) = f(d, i_d + 1)$. For example, $wt((1,1),(2,1))$ would be the fanout $f(A,2)$ of dimension A at level 2. We say $\vec{v}$ is a *successor* of $\vec{u}$, provided $\exists d : 1 \leq d \leq k$ such that $\vec{v}$ is a *d-successor* of $\vec{u}$. As usual, $\vec{u} < \vec{v}$ abbreviates $u \leq v \ \&\ u \neq v$. Figure 3 shows the query class lattice associated with the star schema of Figure 1. In that figure, $(0,0)$ (resp., $(2,2)$) is the least (resp., greatest) element, and $(1,0)$ (resp., $(0,1)$) is the A-successor (resp., B-successor) of $(0,0)$.

Definition 3 [Monotone Lattice Path] A *monotone lattice path* is a sequence of points $(\vec{u}_1, ..., \vec{u}_t)$ such that: (i) $\vec{u}_1 = \perp$, (ii) $\vec{u}_t = \top$, and (iii) $\vec{u}_{i+1}$ is a successor of $\vec{u}_i$, $1 \leq i \leq t-1$. ■

Each monotone lattice path corresponds to a clustering strategy, as illustrated in Section 2. Each edge $(\vec{u}, \vec{v})$ in a lattice path, where $\vec{u} = (i_1, ..., i_d, ..., i_k)$ and $\vec{v} = (i_1, ..., i_d + 1, ..., i_k)$, specifies one loop over sibling entries at level i_d of dimension d. Loops are specified from inner-most to outer-most, and when executed result in a linear ordering of all the cells in the corresponding k-dimensional grid. In the sequel, a lattice path shall mean a monotone lattice path.

Example 2 [Lattice Paths]
Strategy P_1 in Example 1 corresponds to the lattice path $\langle \perp = (0,0), (0,1), (0,2), (1,2), (2,2) = \top \rangle$, whereas strategy P_2 corresponds to
$\langle \perp = (0,0), (0,1), (1,1), (1,2), (2,2) = \top \rangle$. ■

Clearly, there is no benefit to distributing records belonging to a single cell (that is, matching on all k dimensional attributes). Therefore, the clustering problem is one of selecting a linearization order over the cells in a multi-dimensional space. For analytical purposes, the number of continuous fragments of this linearization curve required to cover all cells selected by a query Q is known to serve as a pretty good surrogate for the expected I/O cost of query Q. This is standard in previous literature on clustering.

An arbitrary clustering strategy S (not necessarily a lattice path) is represented as a path passing through all the cells in the k-dimensional data grid associated with the fact table F. We classify each edge on this path as *diagonal* or *non-diagonal*: an edge is *non-diagonal* if the two cells connected by this edge differ in only one dimension. Otherwise an edge is *diagonal*. A clustering strategy is called *diagonal* if it includes at least one diagonal edge. Otherwise it is *non-diagonal*. Clearly, the row-wise strategy, corresponding to the lattice path P_1 (Example 2) in Figure 1, is diagonal. The Hilbert ordering is an example of a non-diagonal clustering strategy.

We can go further and define the *type* of an edge more precisely as the set of dimensions in which the

cells at its end-points differ. Each non-diagonal edge is aligned along exactly one of the dimensions, so there are exactly k non-diagonal edge types. However, there are a combinatorial number of diagonal edge types.

Definition 4 [Characteristic Vector] Let **S** be a k-dimensional star schema and $\mathcal{T} = \tau_1, ..., \tau_t$ be any fixed enumeration of the set of all possible edge types over the k-dimensional grid associated with **S**. Let S be any clustering strategy. Then the *characteristic vector* (CV) of S is defined as $\mathcal{CV}(S) = (e_1, ..., e_t)$, where e_i is the number of edges of type τ_i that are on S. ■

For two dimensions with complete n-level binary hierarchies on either dimension, we denote the CV as $(a_1, ..., a_n; b_1, ..., b_n; d_{11}, d_{12}, ..., d_{1n}, ..., d_{nn})$, which we sometimes abbreviate to $(\vec{a}; \vec{b}; \vec{d})$. For example, Strategy P_1 of Figure 1 has the CV, $\mathcal{CV}(P_1) = (8, 4; 0, 0; 0, 2; 0, 1)$, meaning it has 8 (resp., 4) edges of type A_1 (resp., A_2), 2 (resp., 1) diagonal edges of type D_{12} (resp., D_{22}), and 0 edges of all other types. The Hilbert strategy (Figure 2(b)) has the CV $\mathcal{CV}(H_d^2) = (6, 1; 6, 2; 0, 0; 0, 0)$. Note the 0 diagonal edges and the almost even distribution of edges between corresponding levels of the two dimensions. We often truncate CVs with 0 diagonal entries by dropping all diagonal entries, as in $\mathcal{CV}(H_d^2) = (6, 1; 6, 2)$.

4 Optimal Lattice Path

Let $R = \langle \vec{u}_1, ..., \vec{u}_s \rangle$ be any path in a lattice, $\mathcal{L}$, such that $\vec{u}_{i+1}$ is a successor of $\vec{u}_i$, $1 \leq i \leq s-1$ (it need not connect $\perp$ and $\top$). Define the *length* of R as the product of the weights of edges on R, i.e., $len(R) = \prod_{1 \leq i \leq s-1} wt(\vec{u}_i, \vec{u}_{i+1})$. We define the length of an empty path from a point to itself as 1. Let P be a monotone lattice path and $\vec{u}$ be a point in $\mathcal{L}$ that is not on P. Then the *distance* between $\vec{u}$ and P is defined as $dist_P(\vec{u}) = min\{len(R) \mid R$ is a monotone path segment between $\vec{u}$ and some point on $P\}$. Intuitively, $dist_P(\vec{u})$ corresponds to the average cost of the clustering strategy P on a typical query in the class $\vec{u}$. E.g., in Figure 3, $dist_{P_1}(0,1) = 1$, while $dist_{P_1}(2,0) = 2 \times 2 = 4$. Let μ be any workload such that $\mu(\vec{v}) = p_{\vec{v}}$, for each query class $\vec{v}$. We then define the *(expected) cost* of the monotone path P as the sum of its distances from all points in the lattice, weighted by the probability of each point in the given workload. That is, $cost_\mu(P) = \sum_{\vec{u} \in \mathcal{L}} (p_{\vec{u}} \times dist_P(\vec{u}))$. This is the expected cost of the clustering strategy P over the given workload μ. We wish to minimize this cost.

We next develop a dynamic programming algorithm for finding the cheapest path. We begin with the following key lemma. For a point $\vec{u} \in \mathcal{L}$, we define the *sublattice rooted at* $\vec{u}$ as $\mathcal{L}_{\vec{u}} = \{\vec{v} \mid \vec{v} \in \mathcal{L}$ & $\vec{u} \leq \vec{v}\}$. E.g., in Figure 3, $\mathcal{L}_{(1,1)}$ is the diamond consisting of the points $\{(1,1), (2,1), (1,2), (2,2)\}$.

Lemma 1 [Principle of Optimality]: *If a path $P = (\vec{v}_1, \vec{v}_2, ..., \vec{v}_t = \top)$ is an optimal path w.r.t. workload μ in $\mathcal{L}_{\vec{v}_1}$, then the segment $R = (\vec{v}_2, ..., \vec{v}_t)$ is necessarily optimal w.r.t. μ in $\mathcal{L}_{\vec{v}_2}$.* ■

The proof follows from the additive nature of the cost metric and Bellman's principle of optimality. We omit it here. For a given workload μ, we use P_μ^{opt} to denote the optimal lattice path w.r.t. μ. Figure 4 shows an algorithm, for the two-dimensional case, for finding the optimal lattice path as well as its expected cost over a given workload. The following theorem establishes the correctness and complexity of this algorithm.

Theorem 1 [Optimal Lattice Path]: *Algorithm Find-Optimal-Lattice-Path finds the optimal lattice path over a given workload as well as its expected cost correctly. It takes time linear in the size of the query class lattice.*

Proof. The correctness of the algorithm follows from the following observations, and Lemma 1:

1. For a query class (i,j), if the optimal path from (i,j) to (m,n) passes through $(i, j+1)$, then its expected cost for the set of query classes $(i', j), i \leq i' \leq m$ is $p_{i,j} + f(A, i+1) \times p_{i+1,j} + \cdots + f(A,m) \times \cdots \times f(A, i+1) \times p_{m,j}$, which is equal to $raw_B(i,j)$. A similar remark holds w.r.t. $raw_A(i,j)$.

2. The cost of the optimal path from (i,j) to (m,n) is the minimum between $cost_\mu(i+1, j) + raw_A(i,j)$ and $cost_\mu(i, j+1) + raw_B(i,j)$.

3. Finally, the problem is completely characterized by the following recurrences:

$$cost_\mu(m,n) = p_{m,n}. \quad (1)$$

$$raw_A(i,n) = p_{i,n}, \quad 0 \leq i \leq m. \quad (2)$$

$$raw_B(m,j) = p_{m,j}, \quad 0 \leq j \leq n. \quad (3)$$

$$raw_A(i,j) = p_{i,j} + f(B, j+1) \times raw_A(i, j+1), \quad 0 \leq i \leq m,\ 0 \leq j \leq n-1. \quad (4)$$

$$raw_B(i,j) = p_{i,j} + f(A, i+1) \times raw_B(i+1, j), \quad 0 \leq i \leq m-1,\ 0 \leq j \leq n. \quad (5)$$

$$cost_\mu(m,j) = p_{m,j} + cost_\mu(m, j+1), \quad 0 \leq j \leq n-1. \quad (6)$$

$$cost_\mu(i,n) = p_{i,n} + cost_\mu(i+1, n), \quad 0 \leq i \leq m-1. \quad (7)$$

$$cost_\mu(i,j) = min\{cost_\mu(i+1, j) + raw_A(i,j), cost_\mu(i, j+1) + raw_B(i,j)\}, \quad i < m,\ j < n. \quad (8)$$

Computing each $raw_A()$ (as also $raw_B()$) entry requires 1 addition and (at most) 1 multiplication. Thus,

```
Algorithm Find-Optimal-Lattice-Path(f, m, n, μ);
//μ – workload;
//f(A,i) = fanout of dimension A at level i, etc.;
//m (n) = number of levels in dimension A (B).
//output: cost_μ(0,0) – optimal cost;
//opt_path(0,0) – optimal path.
{
   cost_μ(m,n) = p_{m,n};
   opt_path(m,n) = ⟨(m,n)⟩;
   for (i = m; i ≥ 0; i − −)   raw_A(i,n) = p_{i,n};
   for (j = n; j ≥ 0; j − −)   raw_B(m,j) = p_{m,j};
   for (j = n; j ≥ 0; j − −) {
      for (i = m; i ≥ 1; i − −)
         raw_B(i − 1,j) = p_{i−1,j} + f(A,i) × raw_B(i,j);
   }
   for (i = m; i ≥ 0; i − −) {
      for (j = n; j ≥ 1; j − −)
         raw_A(i,j − 1) = p_{i,j−1} + f(B,j) × raw_A(i,j);
   }
   for (i = m; i ≥ 1; i − −) {
      cost_μ(i − 1,n) = p_{i−1,n} + cost_μ(i,n);
      opt_path(i − 1,n) = (i − 1,n) · opt_path(i,n)
   }
   for (j = n; j ≥ 1; j − −) {
      cost_μ(m,j − 1) = p_{m,j−1} + cost_μ(m,j);
      opt_path(m,j − 1) = (m,j − 1) · opt_path(m,j)
   }
   for (i = m − 1; i ≥ 0; i − −) {
      for (j = n − 1; j ≥ 0; j − −) {
         if (cost_μ(i + 1,j) + raw_A(i,j) <
               cost_μ(i,j + 1) + raw_B(i,j)) {
            opt_path(i,j) = (i,j) · opt_path(i + 1,j);
            cost_μ(i,j) = cost_μ(i + 1,j) + raw_A(i,j)
         }
         else {
            opt_path(i,j) = (i,j) · opt_path(i,j + 1);
            cost_μ(i,j) = cost_μ(i,j + 1) + raw_B(i,j);
         }
      }
   }
}
```

Figure 4: Finding the Optimal Lattice Path

computing all $raw_A()$ and $raw_B()$ entries requires a total of $2(m+1)(n+1)$ additions and $2(m+1)(n+1)$ multiplications. Computing each $cost_\mu()$ entry requires at most 2 additions and at most 1 comparison. Thus, computing all $cost_\mu()$ entries requires a total of $2(m+1)(n+1)$ additions. The overall complexity is thus $4(m+1)(n+1)$ additions, $2(m+1)(n+1)$ multiplications, and $(m+1)(n+1)$ comparisons, which is clearly linear in the query lattice size. ■

A naive examination of the search space has complexity $(2^{m+n})!$ for a two-dimensional star schema with complete binary hierarchies of m and n levels on the two dimensions. This is doubly exponential in the total number of hierarchy levels. In contrast, our dynamic programming algorithm finds the optimal lattice path in time $O((m+1)(n+1))$.

Extension of the algorithm for finding the optimal path in $k > 2$ dimensions is conceptually simple, and has been implemented by us. The interested reader is referred to [14] for further details.

4.1 Unbalanced Hierarchies

In all our discussion so far, we have assumed that the hierarchy on each dimension is balanced, in that the length of the path from every leaf to the root is the same. This is indeed often the case in data warehousing, so such a requirement may not prove to be too restrictive.

However, we do not need to make this assumption. Instead, we can simply add dummy intermediate nodes (with one parent and one child each) at *any* levels of the hierarchy as necessary, to create an extended hierarchy that is balanced. This extended hierarchy now has a clearly defined concept of levels. Some fanouts will be 1, but that creates no problems since our algorithm presented above works with the level-wise average fanout.

5 Improvement by Snaking

Recall (from Section 3) that each lattice path leads to clustering strategy as follows. Each edge $(\vec{u}, \vec{v})$ in a lattice path, where $\vec{u} = (i_1, ..., i_d, ..., i_k)$ and $\vec{v} = (i_1, ..., i_d+1, ..., i_k)$, specifies one loop over sibling entries at level i_d of dimension d. Loops are specified from inner-most to outer-most, and when executed result in a linear ordering of all the cells in the corresponding k-dimensional grid. We can obtain a *snaked* clustering from the same lattice path, by simply reversing the direction of the loop index each time we traverse any loop. We call the resulting clustering a *snaked lattice path*. This is formalized below.

Definition 5 [Snaking] Let $P = \langle \bot = \vec{u}_1, ..., \vec{u}_t = \top \rangle$ be a lattice path. Then the *snaked lattice path* corresponding to P, denoted $\widetilde{P}$, is obtained by reversing the clustering order of alternate $\vec{u}_i$ queries in the data grid, for each $i = 2, ..., t-1$. ■

Note that the snaking is applied to the clustering order and not really to the lattice path itself. We abuse terminology for convenience. Snaking can be applied to any lattice path and will never increase its expected cost, no matter what the workload, and on most workloads, will reduce its expected cost. The intuition is that a snaked lattice path has no diagonal edges. Figure 5 illustrates the idea of snaking.

From this point onward in this paper, we restrict our presentation to a two-dimensional star schema where each dimension has an n level hierarchy (i.e., the top-most level is n and the bottom-most level is 0) with a fanout of 2 at each level. The resulting data grid is a square with 2^n rows and 2^n columns.

The astute reader will see how to extend our arguments to the more general case. We have the proofs

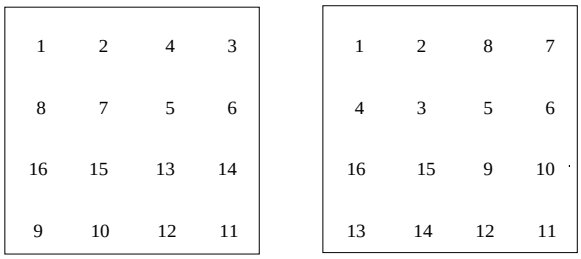

Figure 5: Lattice Paths P_1 and P_2 from Example 1 with Snaking Applied: (a) $\widetilde{P_1}$ and (b) $\widetilde{P_2}$.

for the general case for several of the results established below. The proofs involve considerable notation, and are not presented here.

5.1 Optimality of Snaked Lattice Paths

We wish to compare snaked lattice paths against arbitrary clustering strategies. For this purpose, we shall use the notion of characteristic vector, defined in Section 3. Recall the notation for CVs as well as the notion of edge types from Section 3. We begin with a lemma establishing some properties of such vectors.

Lemma 2 [Constraints on Characteristic Vectors]: *All clustering strategies (on a two-dimensional grid with n-level complete binary hierarchies on either dimension) must satisfy the following constraints.*

$$\begin{aligned}
a_1 &\leq 2^{2n-1}\\
a_1+a_2 &\leq 2^{2n-1}+2^{2n-2}\\
\cdots &\leq \cdots\\
\Sigma_{i=1}^{n} a_i &\leq \Sigma_{i=1}^{n} 2^{2n-i}\\
b_1 &\leq 2^{2n-1}\\
b_1+b_2 &\leq 2^{2n-1}+2^{2n-2}\\
\cdots &\leq \cdots\\
\Sigma_{i=1}^{n} b_i &\leq \Sigma_{i=1}^{n} 2^{2n-i}\\
a_1+b_1+d_{11} &\leq 2^{2n-1}+2^{2n-2}\\
a_1+b_1+b_2+d_{11}+d_{12} &\leq 2^{2n-1}+2^{2n-2}+2^{2n-3}\\
\cdots &\leq \cdots\\
a_1+\Sigma_{i=1}^{n} b_i+\Sigma_{i=1}^{n} d_{1i} &\leq \Sigma_{i=1}^{n+1} 2^{2n-i}\\
\cdots &\leq \cdots\\
\Sigma_{i=1}^{n} a_i+b_1+\Sigma_{i=1}^{n} d_{i1} &\leq \Sigma_{i=1}^{n+1} 2^{2n-i}\\
\Sigma_{i=1}^{n} a_i+b_1+b_2+\Sigma_{(i,j)=(1,1)}^{(n,2)} d_{ij} &\leq \Sigma_{i=1}^{n+2} 2^{2n-i}\\
\cdots &\leq \cdots\\
\Sigma_{i=1}^{n} a_i+\Sigma_{i=1}^{n} b_i+\Sigma_{(i,j)=(1,1)}^{(n,n)} d_{ij} &= \Sigma_{i=1}^{2n} 2^{2n-i}
\end{aligned}$$

Proof Sketch. The last constraint (the only equality) comes from the fact that the number of cells is 2^{2n} so any path passing through all points must have exactly $2^{2n}-1$ edges. For each of the remaining inequalities, the underlying rationale is one of the following: (i) the total number of edges of the type indicated that exist is bounded by the RHS; *or* (ii) if the LHS exceeds the indicated bound, the strategy must contain a cycle, and hence cannot be a clustering strategy. For instance, $a_1 \leq 2^{2n-1}$, because this is the total number of edges of type A_1. As another example, consider $\Sigma_{i=1}^{n} a_i \leq \Sigma_{i=1}^{n} 2^{2n-i}$. Clearly, the RHS $= 2^n(2^n-1)$. This is explained by the fact that there are 2^n columns each of which contains 2^n cells. Thus the total number of edges of type A_i where $1 \leq i \leq n$, cannot be more than $2^n(2^n-1)$. Similarly, $\Sigma_{i=1}^{\ell} a_i + \Sigma_{i=1}^{q} b_i + \Sigma_{(i,j)=(1,1)}^{(\ell,q)} d_{ij} \leq \Sigma_{i=1}^{\ell+q} 2^{2n-i}$ applies because there are $2^{2n-\ell-q}$ queries in the query class (ℓ, q). The only edges that can occur inside each (ℓ, q) grid query are of type A_i for some i: $1 \leq i \leq \ell$, or type B_j, for some j: $1 \leq j \leq q$, or of type D_{ij}, for some i, j: $1 \leq i \leq \ell$, $1 \leq j \leq q$, and the number of such edges per (ℓ, q) query is at most $2^\ell \times 2^q - 1 = 2^{\ell+q} - 1$, thus implying the total number of such edges in the whole grid is no more than $2^{2n-\ell-q}(2^{\ell+q}-1) = 2^{2n-1} + \cdots + 2^{2n-\ell-q}$. ■

Definition 6 [Consistent Characteristic Vector] A characteristic vector is *consistent* if it satisfies the constraints in Lemma 2. ■

Let $\vec{u} = (\vec{a}; \vec{b}; \vec{d}), \vec{v} = (\vec{a'}; \vec{b'}; \vec{d'})$ be any two consistent CVs. We define the partial order, $\vec{u} \preceq \vec{v}$, provided: (i) for some i: $1 \leq i \leq n$, we have $a_j = a'_j$, $\forall j$: $1 \leq j \leq i$, and if $i < n$, then $a_{i+1} > a'_{i+1}$, and (ii) for some q: $1 \leq q \leq n$, we have $b_j = b'_j$, $\forall j$: $1 \leq j \leq q$, and if $q < n$, then $b_{q+1} > b'_{q+1}$. As an example, for $n = 2$, $(8, 4; 2, 1) \preceq (1, 11; 1, 2) \preceq (0, 12; 1, 2)$.[1] We say that $\vec{v}$ is $\preceq$-*minimal* provided for every $\vec{u}$ such that $\vec{u} \preceq \vec{v}$, we have $\vec{u} = \vec{v}$. For example, $(8, 4; 2, 1)$ and $(8, 3; 3, 1)$ are two examples of $\preceq$-minimal vectors. At this time, we do not know whether every consistent vector is indeed the CV of some clustering strategy. However, none of our proofs depends on this detail. For a special class of consistent vectors, however, we have the following result.

Lemma 3 [Characteristic Vector for Snaked Lattice Path]: *Let $\vec{v} = (\vec{a}; \vec{b}; \vec{0})$ be a consistent, non-diagonal, and $\preceq$-minimal vector, such that all entries a_i, b_j are powers of 2. Then $\vec{v}$ must be the* CV *of some snaked lattice path.* ■

We extend the definition of $cost_\mu$ to cover arbitrary[2] consistent vectors $\vec{v} = (\vec{a}; \vec{b}; \vec{d})$.

$$cost_\mu(\vec{v}) = \textstyle\sum_{(i,j)=(0,0)}^{(n,n)} p_{ij} \times (1/2^{2n-i-j}) \times \\ (2^{2n} - (\Sigma_{q=1}^{i} a_q + \Sigma_{\ell=1}^{j} b_\ell + \Sigma_{(q,\ell)=(1,1)}^{(i,j)} d_{ij})).$$

[1] Recall that when all diagonal entries are 0, we drop them all.

[2] Regardless of whether $\vec{v}$ is the CV of some strategy or not.

Lemma 4 [Sub-Optimality of Diagonal Strategies]: *Let S_d be any diagonal strategy. Then there exists a consistent non-diagonal vector $\vec{v}$ such that for every workload μ, $cost_\mu(\vec{v}) \leq cost_\mu(S_d)$.*

Proof Sketch. This is a rather intuitive result. The key idea in our proof is that we can show that any consistent vector $\vec{v}_{in} = (\vec{a}; \vec{b}; \vec{d})$ can be transformed into another $\vec{v}_{out} = (\vec{a}'; \vec{b}'; \vec{d}')$ with the following properties: (i) $a_i \leq a'_i$, $1 \leq i \leq n$; (ii) $b_i \leq b'_i$, $1 \leq i \leq n$; (iii) $d'_{ij} = 0$, $1 \leq i, j \leq n$; and (iv) $a'_i + b'_j = a_i + b_j + d_{ij}$, $\forall i, j$. Now, applying this transformation to $\vec{v}_{in} = \mathcal{CV}(S_d)$, each edge e of type D_{ij} in S_d is covered by a unique edge, say f, which is of type A_i or of type B_j, in $\vec{v}_{out}$. Clearly, f favors every query class that e favors. From the extended definition of $cost_\mu$ for consistent vectors, it then follows that $cost_\mu(\vec{v}_{out}) \leq cost_\mu(\vec{v}_{in}) = cost_\mu(S_d)$.

We exhibit such a transformation by an elegant inductive application of the following pivotal claim, stated only for the base case here, for brevity.

Claim 1: $\exists x_1, y_1 : x_1 + y_1 = d_{11}$ & $\vec{v}_2 =_{def} (a_1 + x_1, ..., a_n; b_1 + y_1, ..., b_n; 0, d_{12}, ..., d_{nn})$ is consistent.

It should be obvious that for any workload μ, $cost_\mu(\vec{v}_2) \leq cost_\mu(\vec{v}_{in})$. Intuitively, we split the d_{11} diagonal edges in $\vec{v}_{in}$ (i.e., S_d) into a_1 edges of type A_1 and b_1 edges of type B_1, while preserving consistency. ■

Theorem 2 [Global Optimality]: *For every workload, there exists a snaked lattice path which achieves the global optimal expected cost. More precisely, $\forall$ workload μ: $\exists$ a snaked lattice path $\widetilde{P}$: $\forall$ strategy S': $cost_\mu(\widetilde{P}) \leq cost_\mu(S')$.*

Proof Sketch: Let μ be a given workload. Choose $\widetilde{P}$ to be any snaked lattice path with the least expected cost $cost_\mu(\widetilde{P})$ among all such paths. Let S' be any other strategy. We need to show $cost_\mu(\widetilde{P}) \leq cost_\mu(S')$. By Lemma 4, we know there is a consistent vector $\vec{v}$ with zero diagonal edges such that $cost_\mu(\vec{v}) \leq cost_\mu(\mathcal{CV}(S')) = cost_\mu(S')$. Thus, it suffices to show that $cost_\mu(\widetilde{P}) \leq cost_\mu(\vec{v})$. This we do by identifying a set of snaked lattice paths $\widetilde{S}_i$, and showing that $cost_\mu(\vec{v})$ cannot be simultaneously (strictly) less than the expected cost $cost_\mu(\widetilde{S}_i)$ of all snaked lattice paths $\widetilde{S}_i$ in the set. We call this *sandwich* construction, because the cost of our subject strategy S' is "sandwiched" between the costs of snaked lattice paths.

Let $\vec{v} = (a_1, ..., a_n; b_1, ..., b_n)$. W.l.o.g. we can assume that $\vec{v}$ is *$\preceq$-minimal*, for it it is not, we can always pick a $\preceq$-minimal vector w such that $\vec{w} \preceq \vec{v}$. If all a_i's and all b_j's are powers of 2, we are done on account of Lemma 3. Otherwise, let i (resp., j) be the smallest integer such that a_i (resp., b_j) is not a power of 2. Now, consider the vectors $\vec{v}_1 = (a_1, ..., a_{i-1}, 2^{2n-i-j}, a_{i+1}, ..., a_n; b_1, ..., b_{j-1}, 2^{2n-i-j+1}, b_{j+1}, ..., b_n)$ and $\vec{v}_2 = (a_1, ..., a_{i-1}, 2^{2n-i-j+1}, a_{i+1}, ..., a_n; b_1, ..., b_{j-1}, 2^{2n-i-j}, b_{j+1}, ..., b_n)$. We can show: (i) both $\vec{v}_1$ and $\vec{v}_2$ are consistent; and (ii) on every workload μ, $cost_\mu(\vec{v}) \geq cost_\mu(\vec{v}_1)$ or $cost_\mu(\vec{v}) \geq cost_\mu(\vec{v}_2)$, and (iii) $\vec{v}_1$ and $\vec{v}_2$ are $\preceq$-minimal. Call $\vec{v}_1$, $\vec{v}_2$ the sandwiching vectors of $\vec{v}$. Now, apply this technique iteratively to *both* $\vec{v}_1$ and $\vec{v}_2$, yielding the sandwiching vectors $\vec{v}_{11}, \vec{v}_{12}$ for $\vec{v}_1$, and similarly the sandwiching vectors $\vec{v}_{21}, \vec{v}_{22}$ for $\vec{v}_2$. Then for any given workload μ, $(cost_\mu(\vec{v}) \geq cost_\mu(\vec{v}_1)) \vee (cost_\mu(\vec{v}) \geq cost_\mu(\vec{v}_2))$ and $(cost_\mu(\vec{v}_i) \geq cost_\mu(\vec{v}_{i1})) \vee (cost_\mu(\vec{v}_i) \geq cost_\mu(\vec{v}_{i2}))$, $i = 1, 2$, which together implies $(cost_\mu(\vec{v}) \geq cost_\mu(\vec{v}_{11})) \vee (cost_\mu(\vec{v}) \geq cost_\mu(\vec{v}_{12})) \vee (cost_\mu(\vec{v}) \geq cost_\mu(\vec{v}_{21})) \vee (cost_\mu(\vec{v}) \geq cost_\mu(\vec{v}_{22}))$. Repeated application of this technique to the resulting set of sandwiching vectors will eventually yield a set of vectors, say $\vec{u}_1, ..., \vec{u}_n$, such that each $\vec{u}_i$ is the characteristic vector of some snaked lattice path, say $\widetilde{S}_i$. It follows from the construction that $cost_\mu(\vec{v}) \geq cost_\mu(\vec{u}_i) = cost_\mu(\widetilde{S}_i)$, for some i: $1 \leq i \leq n$. The theorem follows. ■

The following example illustrates the constructions used in Lemma 4 and Theorem 2.

Example 3 Consider a two-dimensional star schema with a three-level binary hierarchy on each dimension. Consider a diagonal strategy S_d whose characteristic vector is $\vec{v}_{in} = (20, 5, 1; 21, 3, 1; 4, 0, 0, 0, 4, 0, 0, 0, 4)$, where the diagonal entries are listed in the order $d_{11}, ..., d_{13}, ..., d_{33}$. We can find a consistent vector with zero diagonal entries, from $\vec{v}_{in}$, using Lemma 4 as follows. As the reader can verify, one valid way to split $d_{11} = 4$ is $x_1 = 4, y_1 = 0$, which gives $(24, 5, 1; 21, 3, 1; 0, 0, 0, 0, 4, 0, 0, 0, 4)$, a consistent vector. Again, a valid way to split $d_{22} = 4$ is $x_2 = 4$ and $y_2 = 0$, yielding $(24, 9, 1; 21, 3, 1; 0, 0, 0, 0, 0, 0, 0, 0, 4)$. Finally, $d_{33} = 4$ also can be split into $x_3 = 4, y_3 = 0$, giving $\vec{v}_{out} = (24, 9, 5; 21, 3, 1; 0, 0, 0, 0, 0, 0, 0, 0, 0) = (24, 9, 5; 21, 3, 1)$, dropping the all-zero diagonal entries. This is also consistent and has expected cost no more than $\vec{v}_{in}$. Now, $\vec{v}_{out}$ is not $\preceq$-minimal. There are several $\preceq$-minimal vectors $\vec{v}$ such that $\vec{v} \leq \vec{v}_{out}$, one of which is $\vec{u} = (27, 8, 3; 21, 3, 1)$. Now, applying the sandwich construction technique used in the proof of Theorem 2, we have that $\vec{u}$ is sandwiched by $\vec{u}_1 = (32, 8, 3; 16, 3, 1)$ and $\vec{u}_2 = (16, 8, 3; 32, 3, 1)$. But then $\vec{u}_1$ is sandwiched by $\vec{u}_{11} = (32, 8, 2; 16, 4, 1)$ and $\vec{u}_{12} = (32, 8, 4; 16, 2, 1)$ while $\vec{u}_2$ is sandwiched by $\vec{u}_{21} = (16, 8, 2; 32, 4, 1)$ and $\vec{u}_{22} = (16, 8, 4; 32, 2, 1)$. The vectors $\vec{u}_{ij}$, $1 \leq i, j \leq 2$ correspond to snaked lattice paths. The reader is invited to verify that on any workload μ, $cost_\mu(\vec{u})$ cannot be simultaneously less than all the costs $cost_\mu(\vec{u}_{ij})$, for $1 \leq i, j \leq 2$. By piecing the preceding lemmas and theorems together, we can see that for some i, j, $cost_\mu(\vec{u}_{ij}) \leq cost_\mu(S_d)$. The vector $\vec{u}_{ij}$ actually corresponds to a snaked lattice path, $\widetilde{S}$. ■

5.2 The Benefit Due to Snaking

In this subsection, we attempt to quantify the benefit derived from snaking. For this purpose, consider any lattice path P and any query class (i,j) that is not on P. Recall that $dist_P(i,j)$ (defined in Section 4) corresponds to the average cost for a typical (i,j) query under strategy P. We extend this notation to $dist_{\widetilde{P}}(i,j)$, to indicate the average cost of a typical (i,j) query under strategy $\widetilde{P}$. Define the *benefit* to query class (i,j) from snaking as the ratio $ben_P(i,j) = dist_P(i,j)/dist_{\widetilde{P}}(i,j)$. The benefit to a query class represents the amount by which its cost is improved by trading diagonal edges for snake edges. In two dimensions, snake edges correspond to type A_i or type B_j edges. For example, consider the lattice path $P_3 = \langle(0,0),(0,1),(1,1),(2,1),(2,2)\rangle$ and the query class $(2,0)$. $dist_{P_3}(2,0) = 4$, whereas $dist_{\widetilde{P_3}}(2,0) = 10/4$, thus yielding a benefit of $4/(10/4) = 1.6$.

Theorem 3 [Limit on Benefit of Snaking]: *For any workload μ and any lattice path clustering strategy P, let $\widetilde{P}$ be the result of applying snaking to P. Then $cost_\mu(P)/cost_\mu(\widetilde{P}) < 2$.*

Proof. Let P be an arbitrary lattice path and (i,j) any query class. Let (q,ℓ) be the point on P such that $dist_P(i,j)$ is actually the length of the path segment connecting (i,j) and (q,ℓ) in the lattice, as defined in Section 4. In this case, we must have either $q = i$ or $\ell = j$, so assume $\ell = j$ w.l.o.g. Let $\mathcal{CV}(\widetilde{P}) = (a_1,...,a_n;b_1,...,b_n)$. Then $dist_P(i,j) = 2^{i-q}$, and $dist_{\widetilde{P}}(i,j) = (2^{2n} - (\Sigma_{1\le s\le i}a_s + \Sigma_{1\le t\le j}b_t))/2^{2n-i-j}$. $dist_{\widetilde{P}}(i,j)$ is minimized when the number of snake edges, i.e., $(\Sigma_{1\le s\le i}a_s + \Sigma_{1\le t\le j}b_t)$, is maximized. This happens when the successor of $(r,\ell+1)$ on P is $(r+1,\ell+1)$, for all $q \le r < i$. In this case, $dist_{\widetilde{P}}(i,j) = (2^{2n} - (2^{2n-1} + 2^{2n-2} + \cdots + 2^{2n-q-j} + 2^{2n-q-j-2} + 2^{2n-q-j-3} + \cdots + 2^{2n-i-j}))/2^{2n-i-j} = 2^{i-q} - (2^{2n-q-j-2} + 2^{2n-q-j-3} + \cdots + 2^{2n-i-j-1})/2^{2n-i-j} = 2^{i-q} - 2^{2n-i-j-1}(2^{i-q} - 1)/2^{2n-i-j} = 2^{i-q} - (1/2)(2^{i-q}-1)$. Thus, the benefit to class (i,j) w.r.t. the path P is $ben_P(i,j) = 2^{i-q}/(2^{i-q} - (1/2)(2^{i-q}-1)) = 1/(1-(1/2)(1-1/2^{i-q}))$. This benefit is clearly maximized when $i-q$ is maximum, which is the case when $i = n$ and $q = 0$, regardless of j. So, the maximum benefit to (n,j) for any j is $1/(1-(1/2)(1-1/2^n)) = 1/(1/2+1/2^{n+1}) < 2$.

Clearly, for a workload ν, $cost_\nu(P)/cost_\nu(\widetilde{P})$ is maximized when $\nu(n,j) = 1$, and $\nu(x,y) = 0, \forall(x,y) \ne (n,j)$. In this case, $cost_\nu(P)/cost_\nu(\widetilde{P}) = ben_P(n,j) = 1/(1/2+1/2^{n+1}) < 2$. ■

We have the following corollary asserting that the snaked optimal lattice path is not much worse than the optimal snaked lattice path.

Corollary 1 *Let μ be a workload, P_μ^{opt} be an optimal lattice path for μ, and $\widetilde{P_\mu^{opt}}$ be its snaked version. Suppose S is a lattice path distinct from P_μ^{opt} and $\widetilde{S}$ is the optimal snaked lattice path. Then $cost_\mu(\widetilde{P_\mu^{opt}})/cost_\mu(\widetilde{S}) < 2$.* ■

In words, the snaked optimal lattice path never performs more than twice worse than the optimal snaked lattice path, for a two-dimensional grid with complete n-level binary hierarchies. We conjecture that a tighter bound exists. In the proof above, we considered the maximum improvement snaking could provide to any lattice path, and the proof shows that the maximum improvement happens for very poor paths. If one starts with paths that are close to the optimal lattice path clustering, we expect that the improvement factor will be much less than 2.

5.3 A Performance Guarantee

Theorem 3 (see Corollary 1) says that the performance of the snaked optimal lattice path can never be more than twice worse than that of the optimal snaked lattice path. Together with the global optimality result of Theorem 2, this yields that the snaked optimal lattice path is at most twice worse than the global optimum clustering strategy. This is significant, since we can find, in time linear in the query lattice size, a strategy whose expected cost is within a factor of at most 2 of the optimal cost.

6 Experimental Evaluation

To validate the practical usefulness of our analyses and techniques we conducted a number of experiments using the TPC-D benchmark. In this section, we describe our experiments and report our findings.

6.1 Experimental Set-Up

We used the Lineltem table from the TPC-D benchmark as our fact table. This is by a good measure the largest table in the benchmark suite. Together with this, we used the Parts table, with parts themselves being grouped by Manufacturer for the parts dimension, and the Supplier table for the supplier dimension. While there are several time fields in the Lineltem table (namely, ShipDate, CommitDate, ReceiptDate), we picked the ShipDate attribute for the time dimensional hierarchy ShipDate→Month→Year.

In all, we thus had 3 dimensions `parts`, `supplier`, and `time` and a corresponding three dimensional data set with very different fanouts along the three dimensions (e.g., 12 months, 7 years, 5 manufacturers supplying an average of 40 parts, and 10 suppliers). Each cell in this data set was populated with zero or more records. The typical size of a record was 125 Bytes. We used a page size of 8K Bytes.

Once we chose a linearization (i.e., clustering) order, we packed the data along that linear order, splitting cells (but not records) across page boundaries. For any query, we could then count the number of pages retrieved as well as the number of seeks required.

In reporting these costs, for number of pages transferred, we normalized by the minimum number of pages that would need to be transferred, assuming that the data was perfectly clustered for each query. This is simply the total number of bytes required by the records selected divided by the page size, rounded up to the nearest integer. For seeks, the minimum number is just 1 per query, if there is perfect clustering, so no normalization is required.

We studied the TPC-D benchmark queries, and derived the access patterns to the LineItem table based on these queries. For instance, query 5 needs LineItem records selected by year and (supplier) region, with no selection on the parts attribute. Query 9 applies a selection by (supplier) nation, year, and part-type. Enumerating thus, we found that 7 of the 17 different query types defined, used LineItem as the basic fact table, and could potentially be represented as a grid query. (The remaining queries either did not use the LineItem table at all, or did so only after joining it with a large Order table).

We then mapped these 7 query types to appropriate grid query classes, making slight modifications to the queries as needed to fit our choices of dimension hierarchies. We then devised various workloads by altering the proportions of the different classes of queries in our expected query mix. In all we present results below for the following workloads.

6.2 Generation of Workloads

For each dimension, we separately considered three probability distributions, as follows. We divided the total probability of 1: either (a) evenly (e.g., 0.33, 0.33, 0.34 for dimensions with 3 levels, and 0.5, 0.5 for 2-level dimensions), or (b) ramping up: (0.1, 0.3, 0.6) for the three levels, and (0.2, 0.8) for the two levels, or (c) ramping down (0.6, 0.3, 0.1) for the three levels and (0.8, 0.2) for the two levels. By considering all possible combinations of such individual distributions, we generated a total of 27 ($=3 \times 3 \times 3$) workloads, and ran our experiments on all workloads.

6.3 Observed Experimental Results

We measured the cost of the following strategies: (i) optimal lattice path, (ii) snaked optimal lattice path, and (iii) the six possible row major strategies. In reporting the results, for each workload, we only show the cost of the best and worst row major orderings for that workload. Of course, these orderings vary depending on the workload. We ran 27 different workloads, and present a selection of the results observed in Table 4, including workload 5, which was the worst case for our algorithm. In all cases, the number of seeks per query was least for the snaked optimal lattice path, in some cases winning by an order of magnitude. In two cases out of 27, the win was marginal, and the number of blocks read was slightly greater than for the best row major ordering. This is not too surprising since the number of blocks read is only loosely correlated with the number of seeks, which is the metric we optimize for. We also have randomness in the way grid cells are mapped across block boundaries.

Workload	Avg. Normalized Blocks Read (Avg. Number of Seeks Per Query)			
	P_μ^{opt}	$\widetilde{P_\mu^{opt}}$	best row major	worst row major
1	1.53 (8.41)	1.52 (7.71)	2.08 (10.85)	5.28 (39.96)
5	2.22 (5.30)	2.19 (5.10)	1.49 (6.60)	3.98 (22.60)
7	1.24 (4.08)	1.25 (3.73)	1.91 (5.53)	5.25 (52.08)
13	1.70 (4.83)	1.65 (4.75)	1.68 (5.81)	9.94 (40.98)
25	1.74 (4.26)	1.74 (3.83)	1.74 (4.14)	6.34 (31.67)

Table 4: Performance for Different Workloads

Fanout	P_μ^{opt}	$\widetilde{P_\mu^{opt}}$	best row major	worst row major
4	1.45	1.44	1.57	3.84
10	1.42	1.39	1.72	4.39
40	1.24	1.25	1.91	5.25

Table 5: Normalized Blocks Read for Workload 7

Tables 5 and 6 shows the variation in the actual normalized number of blocks read, and the normalized number relative to the snaked optimal lattice path, as the fanout (on the parts dimension) increases. We used the workload (indicated as number 7 in Table 5) corresponding to setting low probabilities in

Fanout	P_μ^{opt}	$\widetilde{P_\mu^{opt}}$	best row major	worst row major
4	1.01	1.00	1.09	2.66
10	1.02	1.00	1.24	3.15
40	0.99	1.00	1.53	4.22

Table 6: Normalized Blocks Read Relative to $\widetilde{P_\mu^{opt}}$ for Workload 7

lower levels of the `time` and `parts` hierarchies and higher probability at the higher levels, while keeping the opposite in the `supplier` dimension. We observe that as the fanout increases, the advantage of the snaked optimal lattice path increases.

7 Related Work

The data intensive nature of OLAP queries and their online response requirements has triggered substantial work on optimizing such queries. Previous approaches to this problem have been based on using materialized views [8, 10, 20, 1], indices [9, 16], and caching [19, 2]. While all these approaches are important to improve the overall query response time of OLAP queries, they do not take into account the order in which cells are laid out on disk. Among these approaches, probably the work closest to ours is that of Deshpande et al. [2]. They partition the multi-dimensional space of an OLAP dataset into regions called *chunks*, taking the dimension hierarchies into account when choosing chunk boundaries. These chunks are then used as a (fine granularity) unit of caching in an OLAP system. Chunks are similar to (though coarser-grained than) our notion of grid cells, and their idea of chunking along hierarchy boundaries in each dimension is based on similar intuitions. Unlike our approach where we determine the optimal ordering of laying out of grid cells on the disk, [2] *always* chooses a row-major ordering to obtain a linearization of chunks. Our algorithms and results can be applied in a straightforward fashion to improve the performance of the chunked file organization described in [2].

A closely related problem is that of organizing multi-dimensional arrays to make their access on secondary and tertiary memory devices fast and efficient, given a workload of access patterns. Sarawagi and Stonebraker [21] consider access patterns specified by the size (in each dimension) of n-dimensional rectangles; hierarchies on the dimensions were not considered in the access patterns. They consider the decomposition of the multi-dimensional array into chunks, and propose some heuristics to determine a *uniform shape* for the chunks that reduces the average number of blocks fetched for a specified access pattern. The chunks are always stored in axis order, and [21] additionally determines a good ordering of the array axes to reduce average seek time, given the access pattern.

It is well-known [11, 18, 22] that there is no good ordering of data points in a multi-dimensional space that will permit arbitrary range queries to be answered efficiently. [22] established, given a uniform distribution of key values, that a k attribute selection on a database with N records has a file access cost of $O(N^{(k-1)/k})$. More recent results have shown similar results independent of any data distribution assumptions.

Nonetheless, it is certainly the case that some orderings are (substantially) worse than others. Several linearizations have been proposed, including the "Z-curve" (or bit interleaving) [17], the Gray-code curve [3, 4], the Hilbert curve [6, 12], and variants (*cf.* [15]). There have been several analyses of the expected performance of many of these linearization curves, including [7, 13]. In [5, 12] it was shown that the Hilbert curve was the best linear ordering for a wide variety of applications.

Our results show that there are many circumstances where snaked lattice path clusterings achieve a much better performance than many of the strategies above, including the Hilbert curve.

8 Conclusions and Future Work

Physical layout of data is crucial to data warehouse performance. We considered the problem of finding optimal ways to cluster records of a fact table in a data warehouse, so as to minimize the expected I/O over a given workload, expressed as a probability distribution over query classes. The search space of this problem is doubly exponential in the total number of hierarchy levels. By exploiting the lattice structure formed by the product of dimensional hierarchy levels, we identified an important subclass of clustering strategies called lattice paths. Lattice path clusterings can be arbitrarily better than the well-known Hilbert curve clustering on some workloads, while it can be more expensive than Hilbert on others. We proposed the notion of snaking, which when applied to a lattice path, always reduces its cost. In the full paper, we show that on a representative class of two-dimensional star schemas with n-level complete binary hierarchies on either dimension, the expected cost of the Hilbert strategy is sandwiched between two fixed snaked lattice paths, on every workload. We also showed that on the same class of schemas, for every workload, there is a snaked lattice path which is globally optimal. While the performance of a snaked optimal lattice path can be worse than that of the optimal snaked lattice path, it cannot be more than twice worse.

We complemented our analyses and validated the practical utility of our techniques with experiments using TPC-D benchmark data.

Acknowledgements

We thank Tom Mitchell for asking thought-provoking questions on how to adapt the design of databases in response to learned workload characteristics. We are grateful to Flip Korn and Nick Koudas for comments on an earlier draft of this paper. Lakshmanan's work was supported in part by a grant from NSERC Canada.

References

[1] E. Baralis, S. Paraboschi, and E. Teniente. Materialized view selection in a multidimensional database. In *Proceedings of the International Conference on Very Large Databases*, 1997.

[2] P. M. Deshpande, K. Ramaswamy, A. Shukla, and J. F. Naughton. Caching multidimensional queries using chunks. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Seattle, WA, 1998.

[3] C. Faloutsos. Multiattribute hashing using gray codes. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Washington, DC, 1985.

[4] C. Faloutsos. Gray codes for partial match and range queries. *IEEE Transactions on Software Engineering*, 1987.

[5] C. Faloutsos and Y. Rong. Spatial access methods using fractals: Algorithms and performance evaluation. Technical Report Tech. Report UMIACS-TR-89-31, University of Maryland, 1990.

[6] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 247–252, 1989.

[7] C. Faloutsos, H. V. Jagadish, and Y. Manolopoulos. Analysis of the n-dimensional quadtree decomposition for arbitrary hyper-rectangles. *IEEE Transactions on Knowledge and Data Engg*, 9(3):373–383, May/June 1997.

[8] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proceedings of the International Conference on Very Large Databases*, 1995.

[9] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for OLAP. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 208–219, 1997.

[10] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 205–216, 1996.

[11] J. M. Hellerstein, E. Koutsoupias, and C. H. Papadimitriou. On the analysis of indexing schemes. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 249–256, 1997.

[12] H. V. Jagadish. Linear clustering of objects with multiple attributes. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 332–342, 1990.

[13] H. V. Jagadish. Analysis of the hilbert curve for representing two-dimensional space. *Information Processing Letters*, 62(1):17–22, April 1997.

[14] H. V. Jagadish, L. V. S. Lakshmanan, and D. Srivastava. Snakes and sandwiches: Optimal clustering strategies for a data warehouse. Technical report, AT&T Labs Research and Concordia University, 1999. (in preparation).

[15] X. Liu. *On the ordering of multi-attribute data in information retrieval systems.* Ph. D. Thesis, University of British Columbia, Canada, 1995.

[16] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 38–49, 1997.

[17] J. A. Orenstein and T. H. Merett. A class of data structures for associative searching. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 181–190, 1984.

[18] V. Samoladas and D. P. Miranker. A lower bound theorem for indexing schemes and its application to multidimensional range queries. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 44–51, 1998.

[19] P. Scheuermann, J. Shim, and R. Vingralek. WATCHMAN: A data warehouse intelligent cache manager. In *Proceedings of the International Conference on Very Large Databases*, 1996.

[20] D. Srivastava, S. Dar, H. V. Jagadish, and A. Y. Levy. Answering queries with aggregation using views. In *Proceedings of the International Conference on Very Large Databases*, pages 318–329, Bombay, India, 1996.

[21] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *ICDE'94*, 1994.

[22] Y. Tanaka. Adaptive segmentation schemes for large relational database machines. In *Proc. 3rd International Conference on Database Machines*, Munich, FRG, 1983.